Project no. IST-033576

XtreemOS

Integrated Project
BUILDING AND PROMOTING A LINUX-BASED OPERATING SYSTEM TO SUPPORT VIRTUAL ORGANIZATIONS FOR NEXT GENERATION GRIDS

Design and Implementation of First Advanced Version of LinuxSSI D2.2.8

Due date of deliverable: November 30th, 2008
Actual submission date: December 10th, 2008

Start date of project: June 1st 2006

Type: Deliverable
WP number: WP2.2
Task number: T2.2.3,T2.2.4,T2.2.5,T2.2.6

Responsible institution: INRIA
Editor & and editor's address: Christine Morin
IRISA/INRIA
Campus de Beaulieu
35042 RENNES Cedex
France

Version 1.0 / Last edited by Christine Morin / December 9, 2008

Project co-funded by the European Commission within the Sixth Framework Programme		
Dissemination Level		
PU	Public	√
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Revision history:

Version	Date	Authors	Institution	Section affected, comments
0.1	09/09/08	Christine Morin	INRIA	Initial template + Outline
0.2	15/10/08	Matthieu Fertré	INRIA	General description of IPC semaphores and message queues
0.3	16/10/08	Matthieu Fertré	INRIA	Description of IPC message queues implementation
0.4	22/10/08	Matthieu Fertré	INRIA	Description of IPC semaphore implementation
0.5	22/10/08	Marko Novak	XLAB	Description of execution of XtreemOS-G services on top of LinuxSSI
0.6	23/10/08	Matthieu Fertré	INRIA	Description of shared objects checkpoint/restart
0.7	23/10/08	Marko Novak	XLAB	Description of LinuxSSI DRMAA implementation
0.8	24/10/08	Pierre Riteau	INRIA	Description of kDFS
0.9	13/11/08	Christine Morin	INRIA	Introduction
0.10	13/11/08	Matthieu Fertré	INRIA	Taking comments from Christine into account
0.11	30/11/08	Christine Morin	INRIA	Conclusion and executive summary
0.12	03/12/08	Christine Morin	INRIA	Last corrections before internal reviewing
0.13	04/12/08	Christine Morin	INRIA	Corrections from Jérôme Robert's review
0.14	05/12/08	Christine Morin	INRIA	Corrections from Ramon's review
0.15	09/12/08	Michael Schoettner	UDUS	Two figures for incremental checkpointing part added
0.16	05/12/08	Christine Morin	INRIA	Final corrections from Ramon's review

Reviewers:
Ramon Nou (BSC) and Jérôme Robert (EADS)

Tasks related to this deliverable:

Task No.	Task description	Partners involved°
T2.2.3	Design and implementation of advanced chekckpoint/restart mechanisms	INRIA*,UDUS
T2.2.4	Design and implementation of advanced reconfiguration mechanisms	INRIA*,NEC
T2.2.5	Design and implementation of LinuxSSI distributed file system advanced feature	INRIA*
T2.2.6	Design and implementation of a customizable scheduler	XLAB*,INRIA

°This task list may not be equivalent to the list of partners contributing as authors to the deliverable
*Task leader

Executive Summary

This document reports on the work done as part of WP2.2 during the last twelve months on the design and development of XtreemOS cluster flavour. LinuxSSI is the heart of the foundation layer for XtreemOS cluster flavour. LinuxSSI leverages Kerrighed single system image cluster operating system developed in open source (http://www.kerrighed.org). Most of our work has focused on the design and implementation of new features in LinuxSSI. Moreover, we have developed the mechanisms needed for having the standard XtreemOS-G layer executed unmodified on top of LinuxSSI. These mechanisms allow the Application Execution Management (AEM) service running on top of LinuxSSI to retrieve the information regarding the running processes regardless of their location in the cluster. Finally, we have developed a set of scripts to integrate XtreemOS cluster flavour in the installation and configuration process used in XtreemOS liveCD for Mandriva Linux distribution.

Our work regarding LinuxSSI new features has focused on three main areas: checkpoint/restart mechanisms, kDFS distributed file system and distributed IPC. Concerning checkpointing, we have integrated support for incremental checkpointing and for enabling checkpoint/restart of multithreaded processes. Regarding kDFS, we have significantly improved its stability and performance. We have also integrated an efficient support for file checkpointing. With regards to IPC, we have developed the mechanisms needed to support cluster-wide message queues and semaphores. Our work on reconfiguration mechanisms to support node eviction and failure has been delayed as it can only be done in collaboration with Kerrighed key developers (who are not all in the XtreemOS consortium) to achieve demonstrable results.

In addition to the improvement of LinuxSSI, we have implemented the DRMAA standard interface for job submission systems on top of the system.

Three releases of LinuxSSI have been produced during the last twelve months (LinuxSSI-0.9-alpha in December 2007, LinuxSSI-0.9-beta in May 2008 and LinuxSSI-0.9-beta3 in September 2008). XtreemOS cluster flavour which integrates all the features described in this report is available on the most recent XtreemOS installation CD. XtreemOS-G services (AEM daemon, XtreemFS client) are able to run on top of LinuxSSI, which also supports the system level VO support mechanisms.

Contents

Chapter 1

Introduction

This document describes the results of the work done with workpackage WP2.2 during the last 12 months of the XtreemOS project (M18-M30) on the cluster flavour of the XtreemOS system.

LinuxSSI is the heart of the foundation layer for XtreemOS cluster flavour. LinuxSSI leverages Kerrighed single system image cluster operating system developed in open source (`http://www.kerrighed.org`). Most of our work as part of WP2.2 focuses on the design and implementation of new features in LinuxSSI. Moreover, we have developed the mechanisms needed for having the standard XtreemOS-G layer executed unmodified on top of LinuxSSI. Finally, we have developed a set of scripts to integrate XtreemOS cluster flavour in the installation and configuration process used in the XtreemOS liveCD for Mandriva Linux distribution. Note that RedFlag has also integrated LinuxSSI in the Asianux Linux distribution but this work which is part of WP4.1 is out the scope of this document (see [11]). Table 1.1 summarizes Kerrighed and LinuxSSI services.

We decided to delay the work planned in Task T2.2.4 *Design and implementation of advanced reconfiguration mechanisms*. This work focuses on the implementation of mechanisms to support hot node addition or eviction in a LinuxSSI cluster and in the long term to support node failures (a node failure should not alter the system instances running on the other cluster nodes). Such mechanisms are twofold. They are comprised of (i) the HotPlug framework, a layer implemented on top of the TIPC communication layer and providing a set of functionalities the other LinuxSSI services rely on to manage their own reconfiguration, and, (ii) service specific mechanisms implemented in callbacks triggered by the HotPlug layer to implement all the actions to deal with the internal state of services. Most of LinuxSSI services (with the exception of kDFS) are in fact Kerrighed services maintained by Kerrighed key developers in the Kerrighed open source community. This means that the XtreemOS consortium should work in close cooperation with the Kerrighed open source community on the reconfiguration mechanisms. Considering Kerrighed roadmap and the fact that Kerrighed core developers plan to revisit the current implementation of the HotPlug service, we found that it was not appropriate to devote much efforts to reconfiguration mechanisms during the last 12 months. Moreover, to obtain significant results (full support of hot node eviction for example), the callbacks for all the Kerrighed services have to be implemented. This work cannot be done by the sole XtreemOS consortium by lack of expertise on some Kerrighed services. It would also need much more manpower than available in WP2.2. The fact that we delay the work planned in Task T2.2.4 does not impact the progress of the work in XtreemOS. In fact, even if large clusters are subject to node failures, the

LinuxSSI Services		
Kerrighed Services		
Developed outside XtreemOS consortium by key developers from Kerrighed open source community	Developed in collaboration with Kerrighed open source community	Developed by XtreemOS consortium
KDDM ProcFS Memory Management FAF Ghost	Linux (with TIPC) HotPlug (hot node addition/eviction) Proc (process group checkpointing) Customizable scheduler	kDFS IPC
Services developed on top of LinuxSSI by XtreemOS consortium		
		DRMAA
		Mechanisms needed to run XtreemOS-G services on top of LinuxSSI
Packages created by by XtreemOS consortium		
		LinuxSSI and Kerrighed packages for Mandriva and RedFlag Linux distributions
XtreemOS contributions for pushing the SSI technology in Linux mainstream development		
		Standalone version of Hotplug and KDDM services with kDFS running on top of them

Table 1.1: Kerrighed and LinuxSSI services

failure rate observed in the moderate size clusters used for testing XtreemOS cluster flavour does not prevent us from performing the needed experiments with the reference applications. The reconfiguration mechanisms being orthogonal to other mechanisms, the latter can be developed and tested even if the system is not currently able to manage hot node evictions and failures (hot node addition is supported in the current release).

Since November 2007, LinuxSSI has been extensively tested within the consortium and we have significantly improved the stability of the system.

In this document, we describe the main features that have been developed in XtreemOS cluster flavour since the November 2007 LinuxSSI release. Chapter 2 describes the mechanisms that allow the Application Execution Management (AEM) service running on top of LinuxSSI to retrieve the information concerning the running processes regardless of their location in the cluster. The new features related to application checkpoint/restart developed as part of Task T2.2.3 *Design and implementation of advanced checkpoint/restart mechanisms* are described in Chapter 3. The standard DRMAA interface for job submission systems has been implemented on top of LinuxSSI. This work carried out in the context of T2.2.6 *Design and implementation of a customizable scheduler* is presented in Chapter 4. Chapter 5 focuses on the new features implemented in kDFS distributed file system (work done in Task T2.2.5 *Design and implementation of LinuxSSI distributed file system advanced features*). Chapter 6 is devoted to the implementation of the distributed management of system V semaphores and message queues in LinuxSSI. This work relates to Task T2.2.9 *Pushing SSI mechanisms into Linux mainstream development*. It should be noted that Deliverable D2.2.9 [10] describes additional work we have done to push the LinuxSSI kernel patches needed to implement the single system image technology into Linux main stream development. Concluding remarks are given in Chapter 7.

Chapter 2

LinuxSSI scheduler: Executing XtreemOS-G services on top of LinuxSSI

2.1 Overview

As a Single System Image (SSI) [7] operating system, LinuxSSI presents a cluster of N computers as a single machine with N CPUs. Due to the nature of SSI, all the programs that execute on a single multiprocessor (i.e. shared multiprocessor (SMP)) machine, should be able to run on LinuxSSI. Unfortunately, some of the LinuxSSI components are not SSI compliant, some of them will probably never be, since this would require major modifications of Linux kernel. As a consequence, some of the applications that work on a single SMP machine have to be modified in some ways in order to be able to run on LinuxSSI.

One of such applications is also XtreemOS Application Execution Manager (AEM) [9]. It is a part of the grid layer of the XtreemOS operating system (XtreemOS-G) and is used for executing applications on the different nodes of the Grid system. The deployment of AEM onto the LinuxSSI is shown in Figure 2.1. Since we are dealing with the SSI operating system we only need to run the AEM's execution manager (ExecMng) on a single LinuxSSI node (we will call this node a "head node"). Only the head node is in charge of receiving the Grid jobs that are sent to ExecMng by AEM's job manager (JobMng). When a Grid job is received, a set of processes is created. All the processes from the set are then distributed among the LinuxSSI nodes by the LinuxSSI scheduler [8].

The component that prevents the ExecMng from being able to execute on LinuxSSI "out of the box" is the Linux's "process events connector" [6]. This is a component in Linux kernel which takes care of reporting fork, exec, id change and exit events to userspace. The userspace daemons then process these events in order to maintain an up-to-date list of all the processes that are currently running on the local system. The AEM service uses the process events connector to get notifications on when a particular process has finished. This information is then used for determining when a particular Grid job has finished (i.e. a Grid job is considered finished when all its processes are finished). The main issue with the process events connector in LinuxSSI relies in the fact that on every cluster node, we are only able to receive local process events. At this point, LinuxSSI is unable to automatically forward events to the head node.

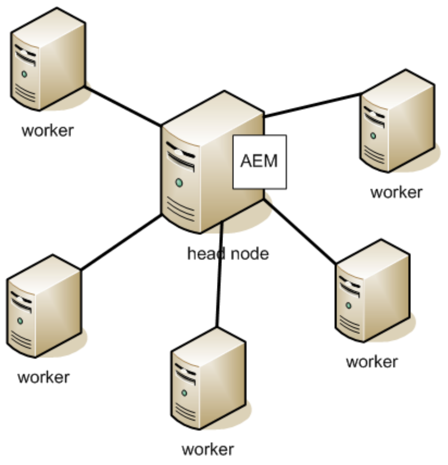

Figure 2.1: Deployment of AEM service onto the LinuxSSI.

To remedy this problem, we decided to implement a simple infrastructure for forwarding process events from the LinuxSSI worker nodes to the head node. This infrastructure consists of two simple daemons (see figure 2.2):

- xos-aem-event-listener: this daemon runs on the LinuxSSI head node. It receives the process events from the workers and passes them to the ExecMng service.
- xos-aem-event-dispatcher: a separate instance of this daemon runs on each worker node. It is in charge of forwarding all the process events that occur on the local machine to the head node.

Both daemons have to be executed on the corresponding nodes when the LinuxSSI cluster is started. After that, they take care of the process event forwarding by themselves.

2.2 Usage

Here is how the process event dispatcher and listener have to be used:

1. the "xos-aem-event-listener" has to be executed on the LinuxSSI head node (i.e. the node on which AEM ExecMng service is running).
2. a separate instance of "xos-aem-even-dispatcher" has to be executed on each worker node.

Below, you can see in which order the daemons have to be executed:

1. First execute "xos-aem-event-listener" on the head node:

```
# ./xos-aem-event-listener --port <port_num>
```

2. Secondly, execute "xos-aem-even-dispatcher" on the rest of the LinuxSSI nodes:

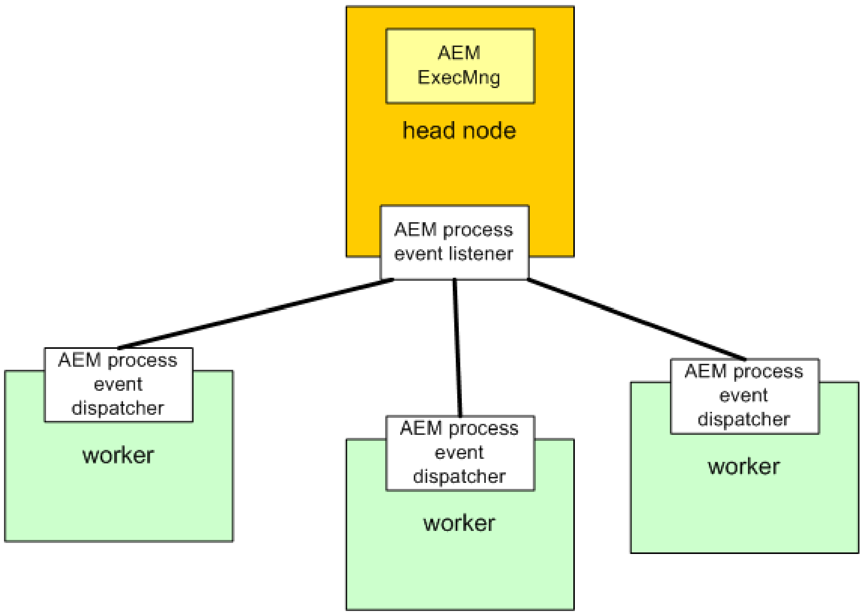
AEM ExecMng
head node
AEM process event listener
AEM process event dispatcher
worker
AEM process event dispatcher
worker
AEM process event dispatcher
worker

Figure 2.2: AEM process event listener and dispatcher.

```
# ./xos-aem-even-dispatcher --ip <IP_address_of_node_A> \
  --port <port_num_of_the_listener>
```

3. At this point you should be able to see some data on the head node every time the process event (e.g. fork, exec, exit, ...) occurs on any worker node.

2.3 Getting the source code and building it

Anybody who would like to modify the listener or dispatcher can retrieve the source for the dispatcher and the listener in the XtreemOS SVN repository: `svn://scm.gforge.inria.fr/svn/xtreemos/foundation/linux-xos-ssi/aem-connector/trunk`. Here you can find the latest stable version.

The building and installation process is very simple. You only execute the following command in the directory where you checked out the sources:

```
# make
# make install
```

The first command builds the listener and dispatcher executables and the second command installs them into the "/usr/local/bin/" directory.

Chapter 3

Checkpointing mechanisms

3.1 Overview

Nodes are susceptible to failures, applications are thus hampered in their execution. In the worst case an application must be restarted from the initial state which means loosing valuable application progress.

Fault tolerance is achieved by checkpoint/restart or rollback recovery - process states are recorded during fault-free execution and recreated after a process/node failure.

In rollback recovery in distributed systems, inter-process dependencies (e.g. communication channels) must be handled. Resolving these dependencies can lead to the domino effect, which can occur when using independent checkpointing strategies. Opposite to that, processes will be synchronized to form a system-wide consistent state in coordinated checkpointing. During communication-induced checkpointing, restart relevant information is piggybacked onto application messages. A consistent job always exists - no domino effect can occur.

LinuxSSI-specific is the usage of the Kerrighed Distributed Data Management (KDDM) service for checkpointing/restarting, as explained in [13]. Over the past year efforts have been spent into consistently saving resources, that are shared by multiple threads and processes. Furthermore, a solution for incremental checkpointing has been developed and integration with the grid checkpointer has been realized.

3.2 Checkpointing multi-threaded Processes

Processes may share different objects with their parent/children, depending on which `clone()` flags were used.

File pointers (`struct file`) opened by a parent process before the fork/clone, are always shared with the resulting child. This is typically the case for `stdin`, `stdout` and `stderr`. It means that the position in the file is shared by the processes and that if one of those processes closes the file, it is closed for the other one too.

Those file pointers are stored in a `struct files_struct` associated to each process. By default, the child gets a copy of the `struct files_struct`of its parent. However if `clone()`

flag CLONE_FILES is used, the 2 processes points to the same struct files_struct, so any file opened (even after the clone) by one process is opened for the other one too.

Using flag CLONE_FS, the struct fs_struct is shared by a parent and its child. This structure is mainly used to store the current working directory. By default, the child gets a copy of the struct fs_struct of its parent.

Other structures may be shared or copied depending on the clone() flags: struct mm_struct, struct signal_struct, struct sighand_struct, struct sem_undo_list. Object struct mm_struct contains all information related to the process address space. Object struct signal_struct keeps track of the shared pending signals. Object struct sighand_struct describes how each signal must be handled by the thread group. Object struct sem_undo_list references all struct sem_undo resulting of operations from the process to System V semaphores.

Thus, checkpointing a tree of processes is not checkpointing each process of the tree independently. When restarting the application, the sharing of objects must be correctly restored to get the expected behavior.

A multi-threaded application is a set of processes that share all those structures and have specific filiation links.

The dump of each object is done only once per application checkpoint. When exporting a process, for each potentially shared object, relevant information is added in a local hashtable. Information about one object should be added only once even if the objects are shared by several processes on the same node. After having exported all application processes, the coordinator checks which objects are shared between several nodes and decides which node is responsible for the checkpoint of a given object. Thanks to information stored in the hashtable, shared objects are dumped.

When the restart of an application begins, after having reserved the orphan session/pgrp pids and before restoring the application processes, we import all shared objects in the context of a fake struct task_struct. A reference to each object (it may be a pointer or an identifier) is added in a local hashtable where we can retrieve it when restoring the application processes.

Implementation has been done for all the different shared objects and checkpoint/restart of multi-threaded application has been successfully tested. A set of tests has been developed and integrated in the KTP Kerrighed test suite based on the standard Linux LTP suite. It is committed in Kerrighed trunk and available in the latest version of LinuxSSI.

3.3 Incremental Checkpointing

3.3.1 Overview

Checkpointing overhead can be reduced by saving only contents that have changed since the last checkpoint. Incremental checkpointing in LinuxSSI is based on a page based granularity.

The major challenge is to detect page modifications by intercepting page-fault exceptions. Generally, a page-fault exception can occur if a program attempts to write on a write-protected page. This exception is detected by the hardware (memory management unit). Afterwards, the page-fault handler can resolve the exception by removing the write-protection. Then, the write bit is set.

Incremental checkpointing in LinuxSSI benefits from this mechanism. At checkpoint time, a modified page can be identified by evaluating the write bit of each process page. If the write bit is set, the page needs to be saved. Over time several incremental checkpoints may be taken. In order to detect further page modifications in subsequent checkpoints, the processor must be enabled to throw a page-fault exception again. Thus, at checkpoint time the write-protection is reactivated by resetting the write-bit per process page (write bit is set to 0).

At restart the last version of the physical process page out of multiple checkpoint images needs to be localized. In LinuxSSI a page table-like structure *inc_cp page table* has been set up that provides a fast lookup (O(1)) to this information at restart, see Figure 3.1. The *inc_cp page table* is hierarchically organised in two levels. In the 32-bit version of LinuxSSI the first 10 bits of a virtual address reference an entry in the first level table, that points to a second level table. The second 10 bits of a virtual address point to an entry in the second level table. This entry contains the file version (e.g. task_mm_pid_vX.bin), in which a page has been saved (the appropriate checkpoint file can be identified). Furthermore, it contains the file offset (multiple pages can be saved in a checkpoint file).

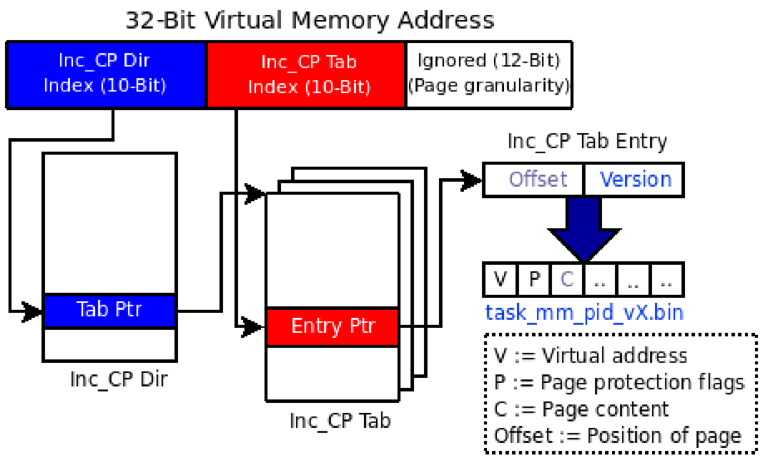
32-Bit Virtual Memory Address
Inc_CP Dir Index (10-Bit)
Inc_CP Tab Index (10-Bit)
Ignored (12-Bit) (Page granularity)
Inc_CP Tab Entry
Offset
Version
V
P
C
..
..
..
task_mm_pid_vX.bin
V := Virtual address
P := Page protection flags
C := Page content
Offset := Position of page
Tab Ptr
Inc_CP Dir
Entry Ptr
Inc_CP Tab

Figure 3.1: Page Control Structure

The *inc_cp page table* is set up, updated and saved to disk for each checkpoint. Each version of the *inc_cp page table* contains localization data for each process' pages. At the first checkpoint, *inc_cp page table* entries for all process pages must be initialized. For subsequent checkpoints, these entries must be updated, that means removing old entries, adding new entries (e.g. new read-only pages) and properly referencing present modified pages.

3.3.2 Special cases

Checking the write-bit for detecting physical address space modifications proved to be insufficient to cover all possible situations, see [1]. The following four cases must be taken into account, too:

- New read-only pages, that were mapped by existing or new memory regions after the initial checkpoint, need to be treated. If there is not yet an *cp_inc page table* entry for the page, it is added, the page is saved.
- An existing memory region gets removed, a new one, covering the same virtual address range, is added in between two incremental checkpoints. Inconsistency occurs during the following

sequence: memory regionA maps fileA, checkpoint, memory regionA is unmapped, memory regionB (covering the same address range) maps fileB, checkpoint, see Figure 3.2. The. If memory regionB is not modified until the next checkpoint, the new content would not be saved by the checkpointer. Thus a monitoring facility working on vma-level (instead of page level), is needed in addition to the page-level access control. A dedicated vma-monitor has been set up, to detect these scenarios and to update the *cp_inc page table* appropriately avoiding the above mentioned problems.

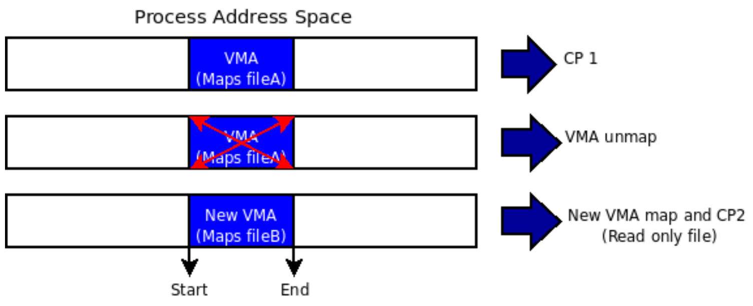

Figure 3.2: VMA Remapping Problem

- An existing memory region gets removed, a new one is added between two incremental checkpoints. This region starts from the same address, but is bigger than the previous one. This case is similar to the previous case. If it is not treated, contents of different application runtimes can get mixed and partially lost. The vma monitor solves this issue, too.
- Removing pages/shrinking vmas must be reflected in the *cp_inc page table* structure. Otherwise it references pages that are not present anymore. The current algorithm covers this scenario.

3.4 Work Directions

The incremental checkpointing functionality must be extended to support 64-bit architectures.

Checkpointing and restart of SYSV IPC shared segments will be reviewed for integrating it into the previously introduced framework for saving and restoring shared structs.

Integration of application callbacks into the LinuxSSI checkpointer is still in progress. It is needed for grid checkpointing (WP3.3). Thus, application callback functions will be executed immediately before checkpointing and after restart.

Chapter 4

DRMAA job submission system

4.1 Overview

One of our main activities during the past year was an implementation of a Distributed Resource Management (DRM) system for LinuxSSI. This is a component which allows applications to submit multiple jobs to a cluster in an easy and standardized way. It also takes care of notifying the application when one of its jobs has finished.

We decided to base our job submission system on the DRMAA (Distributed Resource Management Application API) standard [2]. DRMAA is a specification designed by the Open Grid Forum [5]. It defines all the high level functionality required for applications to submit and manage computational jobs. DRMAA is the most popular standard for job submission systems. It is supported by the majority of the popular systems, such as: Sun Grid Engine (SGE), Condor, Torque/PBS, GridWay, Globus Toolkit and others. It is also very mature (in 2007, DRMAA was one of the two first specifications that reached the full recommendation status in the Open Grid Forum) and applies perfectly to clusters. This makes it suitable for LinuxSSI. The additional benefit of supporting DRMAA standard is the ability to offer the support to all the applications that are based on DRMAA. This effectively means that all the applications that are able to submit jobs to Torque, Condor, SGE and others, should also be able to use LinuxSSI without any modification to their source code.

The scope of the initial DRMAA specification is limited to job submission, job monitoring, and control, as well as retrieval of the finished job status. The interfaces are grouped in five categories:

- init and exit,
- job template handling,
- job submission,
- job monitoring and control,
- auxiliary or informational routines.

The architecture of the LinuxSSI DRM system is shown in Figure 4.1. The LinuxSSI DRM system consists of 2 components:

- DRM server: the server is running on LinuxSSI head node and is in charge of receiving jobs from the clients and creating the processes. The processes are then balanced around the worker nodes by a LinuxSSI scheduler. The server is also in charge of controlling the submitted jobs and notifying the client when the job has finished.
- DRMAA shared library (libdrmaa.so): the LinuxSSI DRMAA library has to be linked to the client application which wants to submit jobs via the DRMAA interface. The library is used for sending different DRMAA requests to the DRM server and for receiving notifications about finished jobs.

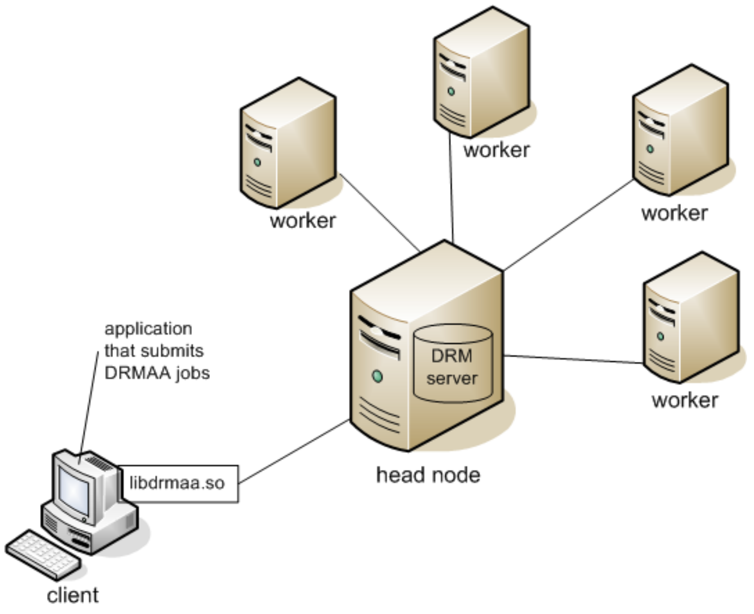

Figure 4.1: Architecture of the LinuxSSI DRM system.

The DRM is also integrated with the LinuxSSI scheduler [8]. As a consequence, all the processes that have been created as a part of a newly received DRMAA job, are immediately taken over by the LinuxSSI scheduler. This scheduler then migrates them to the nodes that are the least loaded. This way, the load is evenly distributed among all the cluster nodes.

4.2 Installation and configuration

The LinuxSSI DRM system should be already installed as a part of XtreemOS Linux distribution if you checked the “LinuxSSI” checkbox in the “Package selection” section of the installation wizard. In case it is not, you have two options:

- either you install it by using the “urpmi” tool of XtreemOS distribution. The below command installs the DRM system automatically:

```
# urpmi krg-drmaa
```

- or you download the source code from the XtreemOS SVN repository, compile it and install it manually. This option is described in detail in Section 4.3.

After you have successfully completed the installation, you should modify the **"krg-drma-server.conf"** configuration file. This file is usually located in the **"/etc"** directory. In doing so, you set the working directory in which all the job directories will be created.

4.3 Building from the source

You also have the possibility to build the DRM server and the DRMAA library by yourself. The source code can be retrieved from the XtreemOS SVN repository:

```
svn+ssh://<username>@scm.gforge.inria.fr/svn/xtreemos/foundation/
linux-xos-ssi/drmaa/trunk.
```

4.3.1 Prerequisites

In order to be able to build the LinuxSSI DRMAA library and the Distributed Resource Management (DRM) server, you need the following libraries installed:

- GLib 2.0, version 2.14.6 or greater
- UUID library (libuuid-dev)

Both libraries should be installed in the XtreemOS distribution by default.

4.3.2 Build process

The installation itself is very simple. You only execute the following commands in the LinuxSSI DRM source code directory:

```
# cd <krg-drmaa-source-directory>
# ./autogen
# ./configure
# make
# make install
```

4.3.3 Executing the Distributed Resource Management (DRM) server

Running the Distributed Resource Management (DRM) server (you need to start this in order to be able to submit jobs to LinuxSSI cluster):

The DRM server is started with the following command:

```
# krg-drmaa-server <-i drm-server-ip> <-p drm-server-port> \
  <-c path-to-config-file>
```

Where the **"drm-server-ip"** is the IP address of the computer the DRM server will be running on (this argument is optional. If the user does not specify it, the DRM server tries to figure out the local IP address automatically), **"drm-server-port"** is the port number the DRM server is listening to for incoming connections and **"path-to-config-file"** is the path to the configuration file (usually this is the "/etc/krg-drmaa-server.conf"). Below you can see the example command for executing LinuxSSI DRM server.

```
# krg-drmaa-server -i 172.16.77.11 -p 1234 \
  -c /etc/krg-drmaa-server.conf
```

Once the server is running, you can start submitting jobs to it. The LinuxSSI DRMAA implementation already contains a couple of test cases that you can invoke to see how the DRM system is working. They are located in the **"tests"** subdirectory of the LinuxSSI DRMAA source tree. For further details on how to retrieve the DRMAA source code from the XtreemOS SVN repository and how to build it, read Section 4.3.

All the tests are invoked the same way:

```
# cd <krg-drmaa-source-directory>/tests
# <test-executable> <drm-server-ip> <drm-server-port>
```

Where **"drm-server-ip"** is the IP address of the DRM server and **"drm-server-port"** is the port number the DRM server is listening to.

For example:

```
# cd <krg-drmaa-source-directory>/tests
# ./test-drmaa-job-run 172.16.77.11 1234
```

4.4 Pointers on developing DRMAA applications

The best reference documentation for developing DRMAA applications is certainly the DRMAA specification [3] itself, and the DRMAA C language binding [4]. They contain detailed descriptions of all the important DRMAA functions for job submission and control. A summarized description of those functions can also be found in the **"drmaa.h"** which is located in the "lib" subdirectory of the LinuxSSI DRMAA source tree.

An even better way for learning how to use DRMAA interface is to examine the source code of the existing applications that use DRMAA. In the "tests" subdirectory of the LinuxSSI DRMAA source tree, we have put a couple of very simple test programs which demonstrate how to submit a job and control it. Below you will find a list of examples along with the explanation on what they do:

- test-drmaa-init.c: demonstrates how to initialize a new DRMAA session and later terminate it,

- test-drmaa-job-template.c: demonstrates how to create a new job template which contains all the necessary information about the job the user wants to submit,
- test-drmaa-job-run.c: demonstrates how to submit job via the DRMAA interface,
- test-drmaa-bulk-jobs-run.c: demonstrates how to submit multiple instances of the same job (i.e. a batch job) by invoking a single DRMAA function. Each job instance invokes the same command, however the parameters can be different,
- test-drmaa-job-ps.c: demonstrates how to retrieve the status of a job that was already submitted to the DRM system,
- test-drmaa-job-wait.c: demonstrates how to use DRMAA interface to wait for the completion of a specific job,
- test-drmaa-job-synchronize.c: demonstrates how to use DRMAA interface to wait for the completion of multiple jobs that were submitted to the DRM system,
- test-drmaa-file-streams.c: demonstrates how to redirect the input to be read from a file instead of stdin and the output to be written to a different file instead to stdout.

4.4.1 Example of DRMAA application: test-drmaa-job-run.c

The source code below is a simple application which demonstrates how to submit a job to LinuxSSI cluster via DRMAA:

```
#include <stdio.h>

#include <drmaa.h>


static int testAttributesSet(drmaa_job_template_t *jt)
{
        char err_buff[DRMAA_ERROR_STRING_BUFFER];
        const char *cmd = "/bin/echo";
        const char *argv[] = {"Hello", "world!", NULL};


        /* add scalar mandatory, optional parameter */
        if (drmaa_set_attribute(jt, DRMAA_REMOTE_COMMAND, cmd,
                err_buff, DRMAA_ERROR_STRING_BUFFER) !=
                DRMAA_ERRNO_SUCCESS) {

                printf("error:␣failed␣to␣set␣attribute␣%s\n",
                        DRMAA_REMOTE_COMMAND);
                return -1;
        }

        if (drmaa_set_vector_attribute(jt, DRMAA_V_ARGV, argv,
                err_buff, DRMAA_ERROR_STRING_BUFFER) !=
                DRMAA_ERRNO_SUCCESS) {

                printf("error:␣failed␣to␣set␣vector␣attribute␣%s\n",
                        DRMAA_V_ARGV);
                return -1;
        }

        return 0;
}

int main(int argc, char *argv[])
{
        drmaa_job_template_t *jt;
        char err_buff[DRMAA_ERROR_STRING_BUFFER];
        char job_id[50];

        char buffer[1024];
        if (argc != 3) {
                printf("Usage:␣test-drmaa-bulk-jubs-run␣"
                        "<drmaa-server-IP>␣"
                        "<drmaa-server-port>\n");
                return -1;
        }

        /* init DRMAA. */
        sprintf(buffer, "ip=%s;␣port=%s", argv[1], argv[2]);
        if (drmaa_init(buffer, err_buff,
                DRMAA_ERROR_STRING_BUFFER) != DRMAA_ERRNO_SUCCESS) {

                printf("error:␣drmaa_init␣failed\n");
                return -1;
        }

        /* create job template. */
        if (drmaa_allocate_job_template(&jt, err_buff,
                DRMAA_ERROR_STRING_BUFFER) != DRMAA_ERRNO_SUCCESS) {

                printf("error:␣failed␣to␣create␣jt1␣job␣template\n");
                return -1;
        }
        /* test setters. */
        if (testAttributesSet(jt) != 0) {
                printf("TEST␣FAILED\n");
                return -1;
        }

        if (drmaa_run_job(job_id, 50, jt, err_buff,
                DRMAA_ERROR_STRING_BUFFER) != DRMAA_ERRNO_SUCCESS) {

                printf("error:␣job␣run␣failed\n");
                printf("TEST␣FAILED\n");
                return -1;
        }

        /* delete job template. */
        if (drmaa_delete_job_template(jt, err_buff,
                DRMAA_ERROR_STRING_BUFFER) != DRMAA_ERRNO_SUCCESS) {

                printf("error:␣failed␣to␣delete␣jt1␣job␣template\n");
                return -1;
        }


        /* exit DRMAA. */
        if (drmaa_exit(err_buff, DRMAA_ERROR_STRING_BUFFER) !=
                DRMAA_ERRNO_SUCCESS) {

                printf("error:␣drmaa_exit␣failed\n");
                return -1;
        }

        printf("TEST␣SUCCEEDED\n");

        return 0;

}
```

From the listing above, we can see the main steps that have to be performed when using DRMAA:

1. DRMAA session initialization: this step needs to be executed before we start using DRMAA functions. It initializes the internal parameters of the DRMAA library and connects to remote DRM server.

2. job template creation: every job is described by a set of attributes (e.g. command that need

to be executed, command arguments, job name, etc.). All the attributes are collected in a job template. In this step, we create a job template for the job we want to submit and set the attributes (the attributes are set in the "testAttributesSet" function).

3. job submission: in this step we submit a newly created job to remote DRM system.

4. destroying DRMAA session: after we have finished, we disconnect from the remote DRM server and clear all the data related with the current DRMAA session.

Chapter 5

kDFS Kernel Distributed File System

5.1 Overview

Traditionally, clusters are designed with a *compute nodes/storage nodes* separation. This architecture can lead to inefficient usage of storage resources: compute nodes do not take advantage of their local devices.

kDFS is a symmetric distributed file system part of Linux-SSI. It is designed to efficiently exploit the storage resources of a cluster. It allows to take advantage of storage resources available on compute nodes, contrary to other file systems which use dedicated storage nodes. Using compute nodes hard drives allows to improve I/O performance by using local resources instead of remote ones.

kDFS also aims at interacting with cluster management services (e.g. process scheduler, checkpoint/restart mechanisms, etc.) to exploit resources more effectively.

kDFS builds on the kernel Distributed Data Manager (kDDM) [12]. kDDM is a distributed shared memory manager at kernel level. It allows kernel services to share data clusterwide, in a coherent manner.

5.2 KDFS core Mechanisms

kDFS builds on the kDDM mechanisms to provide distributed access to a single file namespace shared by all nodes. Both data (file or directory content) and metadata (information about files or directories like size, owner, permissions) are shared via the kDDM mechanisms, using several kDDM sets. One kDDM set is dedicated for storing all metadata (i.e. inode content), and one kDDM is created to share data for each file or directory being accessed.

Data and metadata for files and directories are stored on the hard drives of the compute nodes. The holder of the information is defined when the file or directory is created: data and metadata are stored on the node creating the file or directory. When data is modified by a remote node, the new content is still written on the original node. We plan to improve this behavior by implementing data striping.

During the last 12 months, important improvements were realized on kDFS regarding features, performance and stability. The first new feature is support for file checkpointing.

5.3 Support for file checkpointing

The large number of components in distributed architectures like clusters makes them susceptible to frequent failures. To effectively use these architectures, fault-tolerance mechanisms like checkpoint/restart or rollback recovery must be used (c.f. Chapter 3).

While checkpointing volatile state (registers, virtual memory) and kernel data structures associated with processes can be enough to successfully restart them, processes can be restarted in an incoherent state if they modify files after the checkpoint has been recorded.

It then becomes needed to checkpoint persistent state (file data and metadata) along with the volatile state in order to be able to restart the application in a coherent state.

File checkpointing support was added to kDFS. This feature would enable it to coordinate with the cluster checkpoint/restart mechanism to provide a fully transparent fault-tolerance system.

The file checkpointing implementation is based on a copy-on-write approach. When a page of file data is modified after a checkpoint, it is copied in order to keep both the old and new versions. In the event of a failure, the old version can be used to rollback the file to its state at the time of the checkpoint.

Each checkpoint of a file is stored in a different physical sparse file. This allows to keep a low impact on disk traffic/space usage while making the bookkeeping low as well.

To record in which sparse file each page is stored at a point in time, B-trees of extents are used. Each extent describes in which sparse file a range of contiguous pages of the same checkpoint are stored. Using B-trees allows to efficiently insert, delete and search for extents. There is one B-tree for each file checkpoint. An example of this design is presented in Figure 5.1.

Experiments using the Bonnie++ I/O benchmark showed that even using a high checkpoint frequency results in a very low I/O overhead. Figure 5.2 and 5.3 respectively present write and read performance of Bonnie++ on kDFS using different file checkpoint periods or not checkpointing at all. These two tests show similar I/O throughput whether or not file checkpointing is enabled.

5.4 I/O probes

The scheduling of simultaneously running applications can greatly affect the performance of the system. This is especially true for I/O intensive applications that heavily depend on the behavior of the scheduler (e.g. performance will be poor if the way the scheduler switches to new processes makes the disk do a lot of long seeks).

Making the scheduler aware of the processes resource (CPU, memory, I/O, network) usage enables to schedule applications cleverly, and could provide interesting performance improvements.

In line with the goal of integrating kDFS with cluster services, research was done on how to schedule I/O intensive applications to try to maximize performance.

First, experiments were conducted to determine the impact of simultaneously running different types of applications (heavy CPU users, heavy I/O users, etc.).

With the results obtained from these experiments, work has started on designing and implementing I/O aware scheduling policies using the customizable scheduler framework (see Chapter 2). These policies will use data gathered from I/O probes implemented in kDFS.

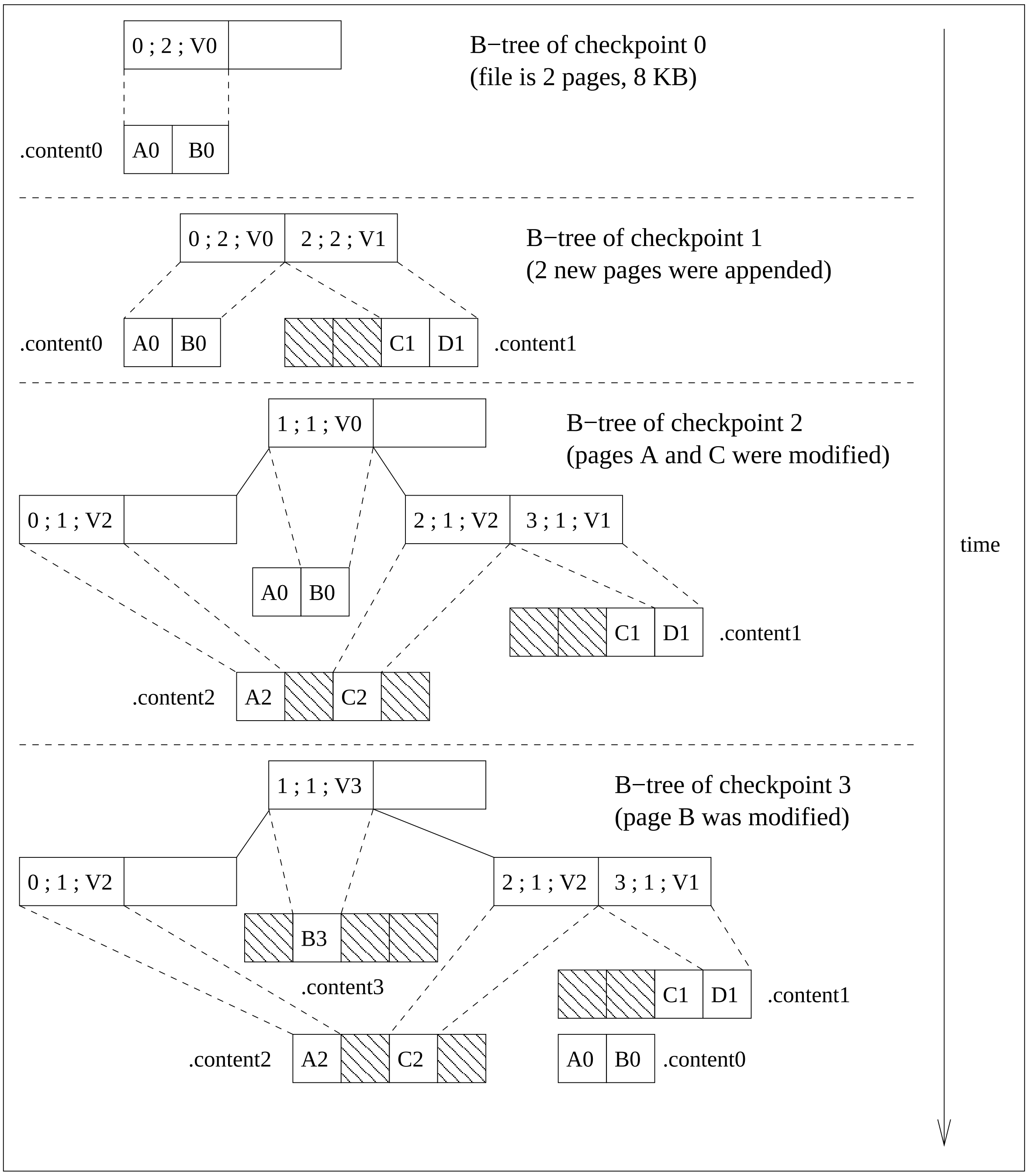

Figure 5.1: Example of extent B-trees and associated sparse files.

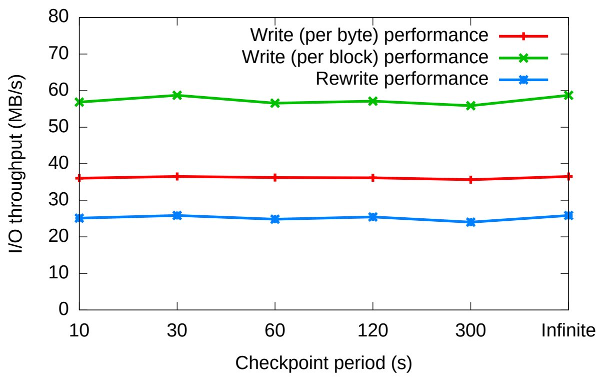

Figure 5.2: Write performance in Bonnie++.
Write performance without checkpoint (infinite period) is compared to performance with several checkpoint periods between 10 and 300 seconds.

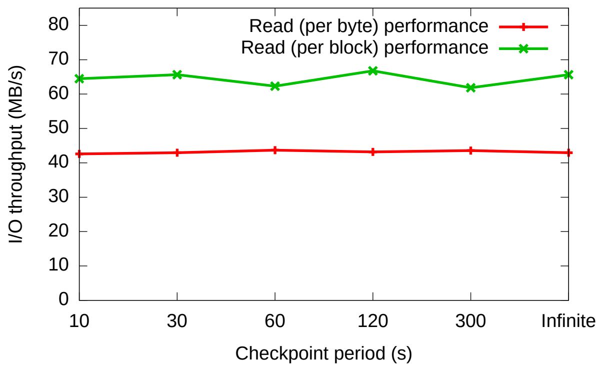

Figure 5.3: Read performance in Bonnie++.
These read operations are performed on files created by the previous write step (i.e. read operations are performed on already versioned files).

5.5 Work directions

First, data striping would create interesting performance improvements by enabling all write operations to be done locally. This would avoid using the network when modifying data in a file initially created on another node. Secondly, research should continue on designing and implementing I/O probes and scheduling policies in order to better utilize available resources.

Chapter 6

Distributed IPC

Inter-Process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes.

System V IPC provides semaphores, message queues and shared memory to handle communication among processes in User Mode on one computer. They have been introduced in AT&T's Unix System V.

The Linux kernel implements these constructs as IPC resources. Just like files, IPC resources are persistent: they must be explicitly deallocated by the creator process, by the current owner, or by a superuser process.

During the last 12 months, we have worked on the support of IPC for processes that run on different nodes of the cluster. Kerrighed was already supporting shared memory segments but was lacking support of semaphores and message queues.

In this chapter, for semaphores and message queues, we first describe the mechanism, then how it is implemented in Linux and which modifications have been done to have full support in LinuxSSI/Kerrighed.

6.1 System V semaphores

6.1.1 General description

Semaphores were invented and proposed by Edsger Dijkstra, and are still used in operating systems today for synchronization purposes. The same mechanism is available for application developers too. It is one of the most important aspects of interprocess communication.

Semaphores are counters used to control access to shared resources for multiple processes. The semaphore value is positive if the protected resource is available and 0 if that resource is busy. This simple mechanism helps in synchronizing multithreaded and multiprocess based applications.

Semaphores basically implement two kinds of operations: a first one that waits on the semaphore value to be positive (the value is then atomically decremented) and another one that increments the semaphore value.

System V semaphores are more complicated than most implementations of semaphore for two main reasons:

- Each IPC semaphore is a set of one or more semaphore values, not just a single value like Posix semaphore. This means that a single IPC semaphore can protect several independent shared resources. You can define one operation for each semaphore value, these operations will be done atomically.
- System V IPC semaphores provide a fail-safe mechanism for situations in which a process dies without being able to undo the operations that it previously issued on a semaphore. When a process uses this mechanism, the resulting operations are called undoable semaphore operations. If the process dies, the values of IPC semaphores are reverted to the values they would have had if the processes had never done any semaphores operations.

6.1.2 Linux implementation

An IPC System V semaphore is represented in the Linux kernel by a `struct sem_array`. This structure can be found either by its IPC key or by its IPC identifier.

As described above, a System V semaphore contains value of several dependent semaphores (`struct sem`) that are allocated just after the `struct sem_array`. For each `struct sem_array`, there is a list (possibly empty) of *pending* set of operations. A set of operations is represented as a `struct sem_queue`. Each operation is to be executed on one particular `struct sem`. A set of operations is pending when all operations can not be executed atomically. Each `struct sem_queue` has a pointer *sleeper* to the process (`struct task_struct`) that has requested this set of operations. At any time, a given process can only be blocked on one System V semaphore.

If the set of operations is marked *undoable*, the `struct sem_queue` has a pointer to a `struct sem_undo`. Like a `struct sem_queue`, it contains a set of operations to execute on a System V semaphore. If the pending process aborts suddenly, this `struct sem_undo` will be used to undo all previous operations done on a given System V semaphore by the aborting process.

Each `struct sem_undo` is indexed in two different lists. The per-process list includes all `struct sem_undo` corresponding to IPC semaphores on which the process has performed undoable operations (clone processes may share the same per-process list, see CLONESYSVSEM in clone() system call manual). The per-semaphore list includes all `struct sem_undo` corresponding to processes that have performed undoable operations on the semaphore.

Figure 6.1 illustrates the relationships between the different data structures implementing System V semaphore.

6.1.3 Kerrighed implementation

In Kerrighed, the `struct sem_array` representing the System V semaphore is stored as a KDDM object [12]. When synchronizing the `struct sem_array` from one node to another, the corresponding list of `struct sem_queue` and per-semaphore `struct sem_undo` list are synchronized too. The synchronization is done thanks to the `semarray_io_linker` KDDM IO Linker.

The biggest challenge in Kerrighed implementation was the management of the double-linked `semundo` list. As explained above, the per-semaphore `struct sem_undo` list is synchronized, so we use the standard Linux implementation to iterate on it. On the contrary, the per-process list

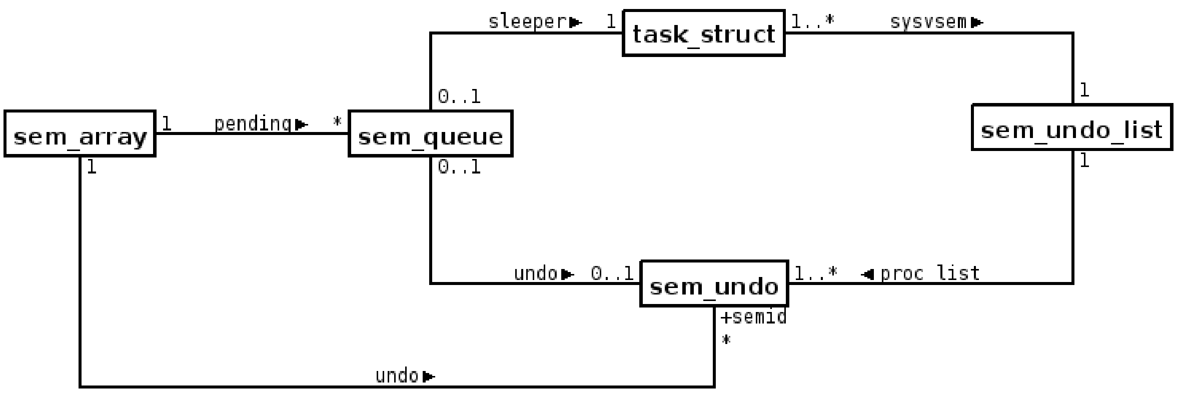

Figure 6.1: Implementation of IPC System V semaphores in Linux

has been modified to use identifiers instead of pointers. Clone processes may share the same per-process `struct sem_undo` list. In Kerrighed, those processes may be hosted on different nodes. Thus, each per-process `struct sem_undo` list is identified thanks to a unique identifier. Thanks to this identifier, we are able to retrieve the list that is stored as a KDDM object thanks to the `semundo_list_io_linker` KDDM IO Linker. The difference is that in this list, instead of having direct access to `struct sem_undo`, we get access to an identifier that finally gives access to the given `struct sem_undo`.

Figure 6.2 illustrates the modifications done on Linux data structures to support distributed management of semaphores.

The Linux System V semaphore code has been adapted to use this design. The choice has been done to modify Linux code instead of coding from scratch because the System V semaphore API is very complicated, and we thought it was easier to adapt the code and then see each case carefully than to rewrite from specifications.

The System V semaphore Kerrighed implementation passes all the related tests from Linux Tests Project.

6.2 System V message queues

6.2.1 General description

Message queues provide an asynchronous communication protocol, meaning that the sender and receiver of the message do not need to interact with the message queue at the same time. Messages placed onto the queue are stored until the recipient retrieves them.

System V message queues are limited in size and in number of messages. Processes that need to send or receive messages from a System V message queue acquire the resource using `msgget()` system call.

Processes exchange messages using `msgsnd()` and `msgrcv()` system calls, which insert a message into a specific message queue and extract a message from it, respectively.

A message is composed of a fixed-size header and a variable-length text. It can be labelled with an integer value (the *message type*), which allows a process to selectively retrieve messages from its

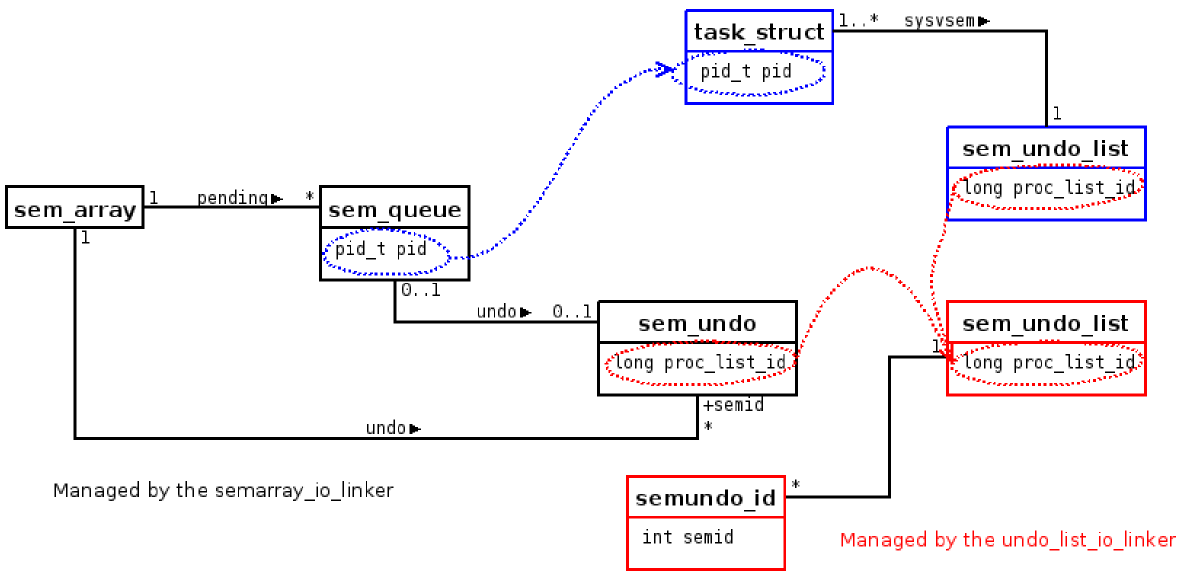

Figure 6.2: Implementation of IPC System V semaphores in LinuxSSI

message queue. Only one process can receive a given message, it is removed from the message queue as soon as one process has received it.

A process may be blocked when sending a message because the queue is full. The process will be unblocked as soon as there is enough place to post the message.

Receiving process may also be blocked when the queue is empty or when the desired type of message is not present in the queue.

As described in the manual page of `msgrcv()` system call, `msgrcv()` needs a *message type* `msgtyp` as argument.

The argument `msgtyp` specifies the type of message requested as follows:

- If `msgtyp` is 0, then the first message in the queue is read.
- If `msgtyp` is greater than 0, then the first message in the queue of type `msgtyp` is read, unless MSG_EXCEPT was specified in `msgflg`, in which case the first message in the queue of type not equal to `msgtyp` will be read.
- If `msgtyp` is less than 0, then the first message in the queue with the lowest type less than or equal to the absolute value of `msgtyp` will be read.

6.2.2 Linux implementation

An IPC message queue is represented in the Linux kernel by `struct msg_queue`. This structure can be found either by its IPC key or by its IPC identifier.

For each `struct msg_queue`, there is a list of messages (possibly empty) `q_messages`. Each message is stored in a `struct msg_msg`. This structure contains the *message type* `m_type` and allows to access the content of the message.

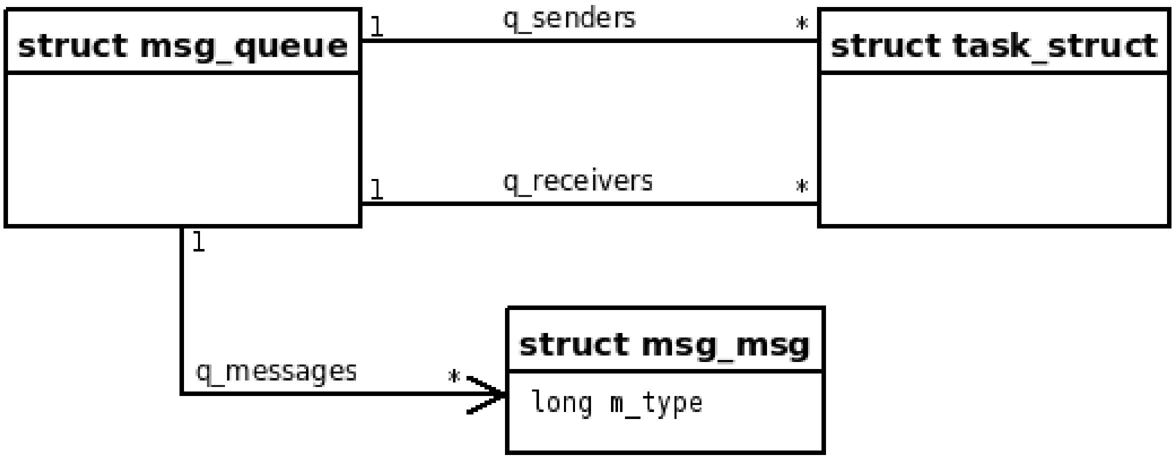

Figure 6.3: Implementation of IPC System V messages queues in Linux

A process (`struct task_struct`) blocked when sending a message because the queue is full is added in the list `q_senders`.

A process blocked when receiving a message is added in the list `q_receivers`.

Figure 6.3 is a UML class diagram showing relationship between the structures.

When a process requests to add a message in the queue, the number of messages and the size of the queues is checked to know if there is enough place to post the message.

When a process requests to receive a message from the queue, each message of the list `q_messages` is checked one by one until a message that matches the desired message type is found.

6.2.3 Kerrighed implementation

In Kerrighed, the `struct msg_queue` representing the message queues is stored as a KDDM object [12].

Basic operations that do not need to access the list of messages are done accessing the `struct msg_queue` through KDDM.

The creator node of the message queue hosts the list of messages. We have added one field (`int is_master`) to the `struct msg_queue` to check if we access it through the master node. The KDDM IO Linker associated to the `struct msg_queue` keeps the whole structure but `is_master` field synchronized on all nodes requesting it.

Send and receive operations from the creation node are done as in Linux kernel. When a send or receive operation is requested from another node, the process sends an RPC request to the creator node. An RPC handler is created, it performs the send or receive operation locally and then sends the results to the requesting process. This design prevents from sending the full list of messages over the network to receive only one or maybe no message.

Figure 6.4 shows an example. Process A requests to receive a message of type 3. It checks the `is_master` value on the corresponding `struct msg_queue`. As it equals to 0 (meaning false), it sends an RPC request to the creation node of the message queue. The RPC handler performs the call that process A has requested, then it sends the result to it.

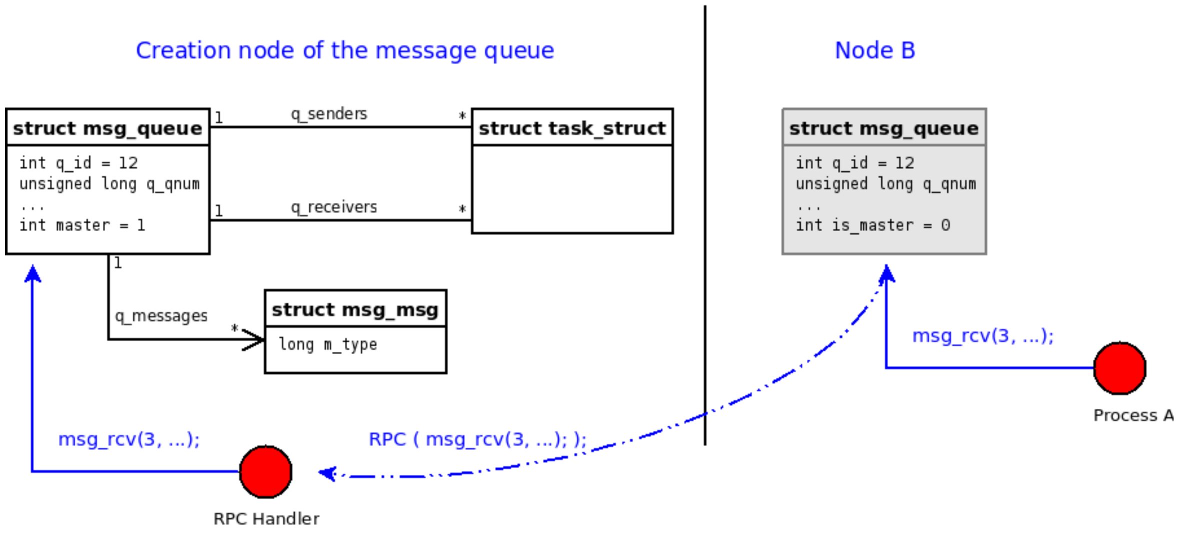

Figure 6.4: Implementation of IPC System V messages queues in LinuxSSI

The System V message queue Kerrighed implementation passes all the related tests from Linux Tests Project. It is integrated in the official Kerrighed and in the latest LinuxSSI versions.

Chapter 7

Conclusion

Three releases of LinuxSSI have been produced during the last twelve months (LinuxSSI-0.9-alpha in December 2007, LinuxSSI-0.9-beta in May 2008 and LinuxSSI-0.9-beta3 in September 2008).

The XtreemOS cluster flavour is available on the latest XtreemOS installation CD. XtreemOS-G services (AEM daemon, XtreemFS client) are able to run on top of LinuxSSI which also supports the system level VO support mechanisms. At the end of November 2008, LinuxSSI provides a stable prototype of kDFS distributed file system able to successfully run standard benchmarks and test suites. Moreover kDFS integrates efficient support for checkpointing and restoring file states. LinuxSSI is now able to checkpoint multithreaded processes and LinuxSSI kernel checkpointer has been optimized through incremental checkpointing. On top of LinuxSSI, the DRMAA standard interface for job submission systems has been implemented and tested. LinuxSSI also provides cluster-wide IPC.

The support for checkpointing/restarting multi-threaded processes, the customizable scheduler and the support for distributed management of IPC have been integrated in the official Kerrighed version.

In our future work directions, we plan to extend the LinuxSSI kernel checkpointer to use the support provided by kDFS for file checkpointing. Moreover, we plan to define and experiment advanced load balancing strategies using the customizable scheduler framework. Depending on the roadmap of Kerrighed community, we may devote efforts to kDFS reconfiguration mechanisms in the coming months. We will continuously provide support to the community and in particular to partners involved in WP4.2 who are in charge of validating XtreemOS and those involved in WP4.1 who are responsible for the packaging of the XtreemOS software.

Bibliography

[1] *Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers*, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Distributed resource management application api (drmaa). `http://www.drmaa.org`, 2008.

[3] Drmaa 1.0 grid recommendation (gfd.133). `http://www.ogf.org/documents/GFD.133.pdf`, 2008.

[4] Drmaa c binding v1.0. `https://forge.gridforum.org/sf/docman/do/downloadDocument/projects.drmaa-wg/docman.root.ggf_13/doc5545`, 2008.

[5] Open grid forum (ogf). `http://www.ogf.org`, 2008.

[6] Process events connector. `http://lwn.net/Articles/153694/`, 2008.

[7] R. Buyya, T. Cortes, and H. Jin. Single system image. *Int. J. High Perform. Comput. Appl.*, 15(2):124–135, 2001.

[8] XtreemOS consortium. Design and implementation of a customizable scheduler. Deliverable D2.2.6, November 2007.

[9] XtreemOS consortium. Design of the architecture for application execution management in xtreemos. Deliverable D3.3.2, June 2007.

[10] XtreemOS consortium. First prototype of the standalone kddm module. Deliverable D2.2.9, November 2008.

[11] XtreemOS consortium. Linuxssi integration and packaging in mandriva and redflag distributions. Deliverable D4.1.5, November 2008.

[12] Adrien Lèbre, Renaud Lottiaux, Erich Focht, and Christine Morin. Reducing Kernel Development Complexity in Distributed Environments. In *14th International Euro-Par Conference (Euro-Par 2008)*, pages 576–586, 2008.

[13] J. Mehnert-Spahn and M. Schoettner. Design and implementation of basic checkpoint/restart mechanism in linuxssi d2.2.3, November 2007.